PERQ
Systems
Corporation

PERQ QCODE REFERENCE MANUAL

March 1984

This manual is for use with POS Release G.5 and subsequent releases until further notice.

PERQ Systems Corporation
2600 Liberty Avenue
P. O. Box 2600
Pittsburgh, PA 15230
(412) 355-0900

The information in this document is subject to change without notice and should not be construed as a commitment by PERQ Systems Corporation. The company assumes no responsibility for any errors that may appear in this document.

PERQ Systems Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

Table of Contents

Table of Contents

1. Q-Machine Architecture

1.A Definitions

Segment - A segment is the underlying structure of PERQ's virtual memory system. It is the largest area of contiguous memory, and also the unit of swappability. Segments come in two types: code segments, which are byte-addressed, read-only, and fixed in size with a maximum size of 64K bytes (32K words); and data segments, which are word-addressed, read-write, and variable in size with a maximum size of 64K words.

MSTACK - Memory Stack. A data segment which contains the user run-time stack.

ESTACK - Expression Stack. A 16 level expression evaluation stack (internal to the PERQ processor).

MTOS - Top of MSTACK. MTOS refers to the virtual address of the top of the memory stack. (MTOS) denotes the item on the top of the MSTACK.

ETOS - Top of ESTACK. (ETOS) denotes the item on the top of the ESTACK.

Activation Record - Stack segment fragment for a single routine containing local variables, parameters, function result, temporaries (anonymous variables), other housekeeping values (Activation Control Block - defined below), and a copy of the EStack at the time the activation record is created.

CB - Code Base (register). Physical adddress of the base of the current code segment.

SB - Stack Base (register). Physical address of the base of the current stack segment.

PC - Program Counter (register). Physical address of the current instruction.

GDB - Global Data Block. A GDB contains the global variables for a particular module. GDBs always begin on a double-word boundary.

ISN - Internal Segment Number (compiler-generated).

SSN - System Segment Number (system-generated). Note, System Segment 0 is reserved and may never be used.

LL - Lexical Level. Note: the Lexical Level of the main body of a process is always 0.

RN - Routine Number (register). RN contains the ordinal number of the current routine. Note: RN must lie in the range 0 to 255.

CS - Code Segment (register). CS contains the system segment number (SSN) for the current code segment. This segment must be resident in physical memory for a process to be runnable.

SS - Stack Segment (register). SS contains the system segment number (SSN) for the current stack segment. This segment must be resident in physical memory for a process to be runnable.

PS - Parameter Size. PS is the number of words in an activation record which are used for parameters.

RPS - Result + Parameter Size. This is the number of words in an activation record which are used for function result and parameters.

LTS - Local + Temporary Size. LTS is the number of words in an activation record which are used for locals and temporaries (anonymous variables). (Note: the LTS of a main program body is always forced to 0.)

AP - Activation Pointer (register). AP contains the physical address of the current activation record.

DL - Dynamic Link. This is the AP of the caller, represented as an offset from SB.

SL - Static Link. This is the AP of the surrounding routine, represented as an offset from SB.

TP - Top Pointer (register). TP contains the physical address of the top of the run-time MStack.

TL - Top Link. TP of the caller, represented as an offset from SB.

GP - Global Pointer (register). Physical address of the GDB for the current code segment.

GL - Global Link. GP of the caller, represented as an offset from SB.

LP - Local Pointer (register). Physical address of the current activation record. When the LP is stored in an Activation Control Block (ACB), it is represented as an offset from SB. Unlike other values in the ACB, the LP value is the current value of the Local Pointer, not some previous value.

XGP - eXternal Global Pointer. Pointer to another code segment's GDB, represented as an offset from SB.

XST - eXternal Segment Table. For a given program module, the XST translates ISNs to SSNs and XGPs.

RS - Return Segment. RS is the CS of the caller.

RA - Return Address. PC of the caller, represented as an offset from CB.

RR - Return Routine. RN of the caller.

RD - Routine Dictionary. Each code segment contains a routine dictionary which is indexed by RN. For each routine, the routine dictionary gives the lexical level (LL), entry address, exit address, parameter size (PS), result + parameter size (RPS), and local + temporary size (LTS).

ACB - Activation Control Block. The ACB contains housekeeping values in the activation record. It contains the SL, LP, DL, GL, RS, RA, RR and EP. In the ACB, the DL, GL, RS, RA, and RR are the AP, GP, CS, PC, and RN of the caller, respectively. The SL is the AP of the routine that surrounds the current one. The LP in the ACB is the current local pointer.

EEB - Exception Enable Block - Each EEB enables a single exception by associating an exception with a handler. A (possibly empty) list of EEBs is associated with each activation record in the stack.

Enabling an Exception - Associating a certain exception handler with a certain exception.

EP - Exception Pointer. The address (as an offset from SB) of a list of nodes that describe which exceptions are enabled in a certain routine.

ER - Exception Routine Number. The routine number of an exception.

ES - Exception Segment Number. The segment number of an exception.

Exception - An error or unusual occurrence in the execution of a routine or program.

Exception Handler - A procedure to be executed when a certain exception is raised.

HR - Handler Routine Number. The routine number of an exception handler.

NE - Next Exception. The address (as an offset from SB) of the next in a list of nodes that describe which exceptions are enabled in a certain routine.

Raising an Exception - Asserting a certain exception.

1.B Memory Organization

The PERQ's virtual memory system features a segmented 32-bit virtual address space mapped into a 20-bit physical address space. The segment is the unit of swappability, and comes in two types:

1) Code segments which are byte-addressed, read-only, and fixed in size with a maximum size of 64K bytes (32K words).

2) Data segments which are word-addressed, read-write, and variable in size with a maximum size of 64K words.

A PERQ process is a collection of up to 64K code and data segments. One of the data segments is the stack segment. Every process must have a stack segment and at least one code segment.

All segments are allocated in 256 word chunks and when in physical memory are aligned on 256 word boundaries. Note: A single segment must exist in contiguous memory. It may not be fragmented.

1.B.1 Memory Organization at the Process Level

The memory organization is designed with the following attributes in mind: 1) to allow separately compiled code segments to be grouped into a single process, 2) to allow code segments to be shared among processes, 3) to allow each code segment to have its own global variables, and 4) to allow one code segment to reference routines and global variables in other code segments. To achieve this, the following high-level characteristics are implemented:

1) All code is re-entrant.

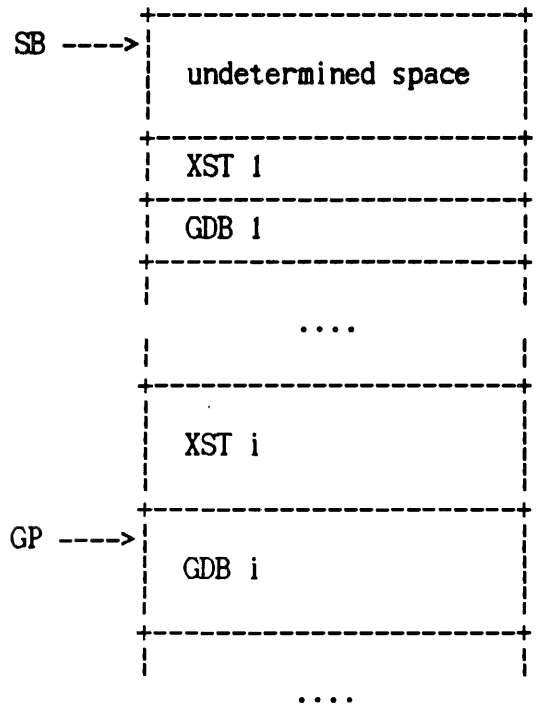
2) Each code segment only refers to other code segments by internal (compiler-generated) segment numbers, which are not necessarily the same as the system-assigned segment numbers.

3) Each code segment in a process has its own Global Data Block on the run-time stack.

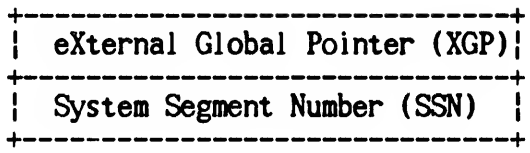
4) Each code segment has an external segment table to permit referencing global variables and routines from other code segments.

1.B.1.a Global Data

At the global level, there is a Global Data Block (GDB) and an eXternal Segment Table (XST) associated with each code segment in a process. For a particular program module, the GDB contains the global variables, and the XST translates internal (compiler-generated) segment numbers (ISNs) to actual system segment numbers (SSNs) and eXternal Global Pointers (XGPs). To simplify the system, we devote a single pointer to reference both the current GDB and XST. This Global Pointer (GP) points to the lowest address in the GDB and is ALWAYS aligned on a double-word boundary.

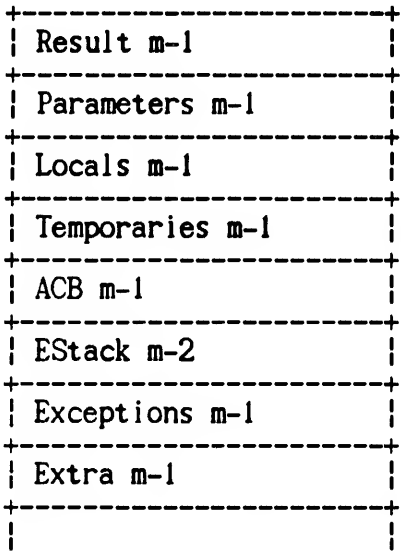


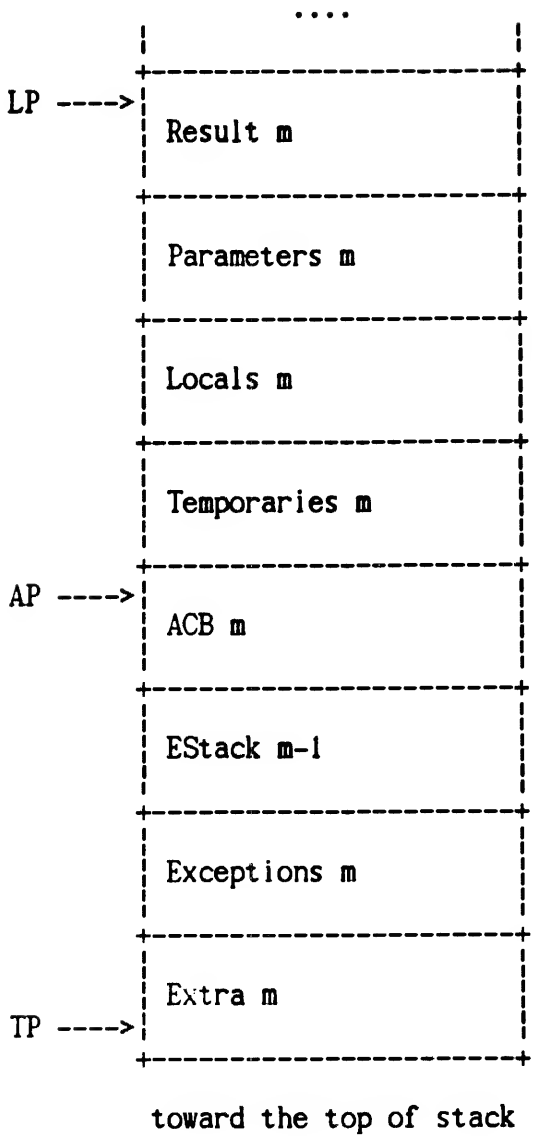
The XST for each segment is indexed by the internal segment numbers (ISNs). The entry is at GP - 2*ISN (Note: There is no entry for ISN 0; ISN 0 always refers to the current segment). Each entry contains the offset from stack base (SB) of an external data block (XGP) and the actual system segment number (SSN) of the external segment. The XGP values are set by the linker, and the SSN values are set by the loader.



1.B.1.b Local Data

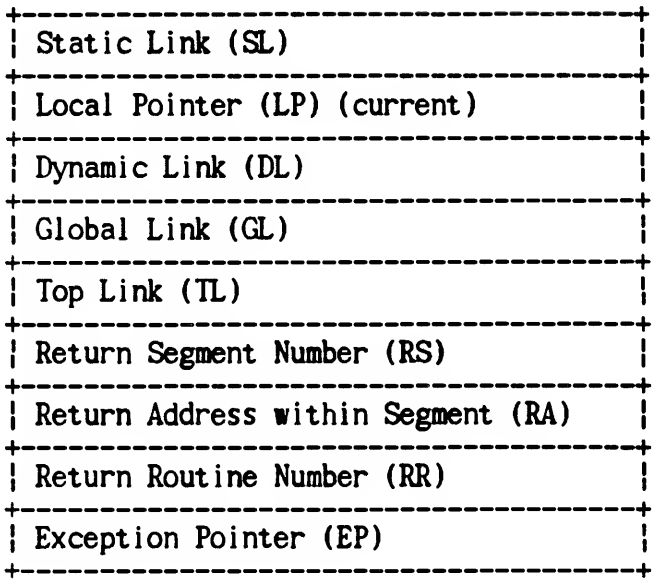
At the local level, there is an activation record, which consists of local variables, function result, parameters, temporaries (anonymous variables), the Activation Control Block (ACB), the previous EStack, exception enable blocks, and extra values that the routine may push and pop from the run-time stack. Three pointers are used to access and keep track of this information: the top-of-stack pointer (TP), the current-activation pointer (AP), and the local-variables pointer (LP).





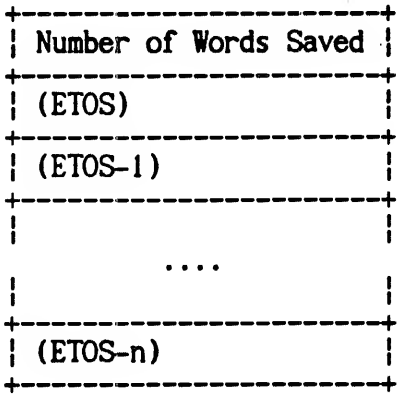
The function result, parameters, locals and temporaries are located by an offset from LP.

Each ACB has the following form:



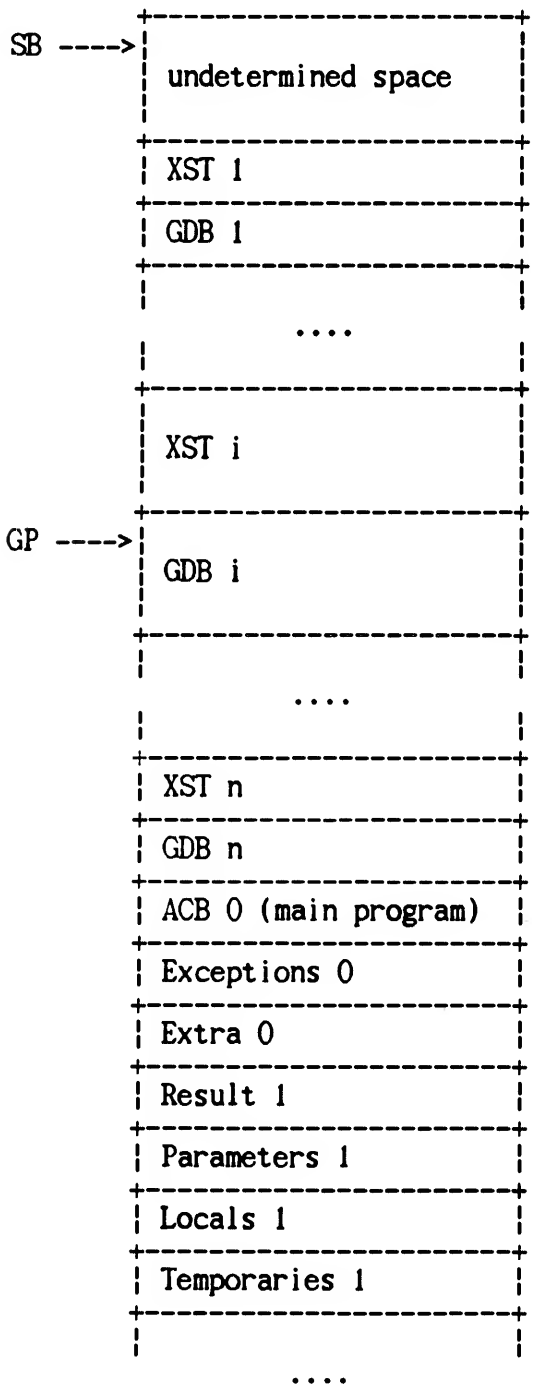
The values in the ACB are the AP of the surrounding routine (SL), the current (not previous) LP, the AP of the caller (DL), the GP of the caller (GL), the TP of the caller (TL), the SSN of the caller (RS), the program counter (PC) of the caller (RA), the RN of the caller (RR) and a pointer to the current exception enable records (EP). Note: When previous pointer values are saved in the ACB they are called links: SL, DL, GL, TL. Because the current (not previous) LP and EP are stored in the ACB, they are called pointers, not links. The static link is not used for main programs and top-level routines. It is zero for these routines and is therefore a means of detecting top-level routines. The dynamic link is zero for the first routine on the stack (the one at the base of the stack). This is used to detect the end of the stack during stack searches.

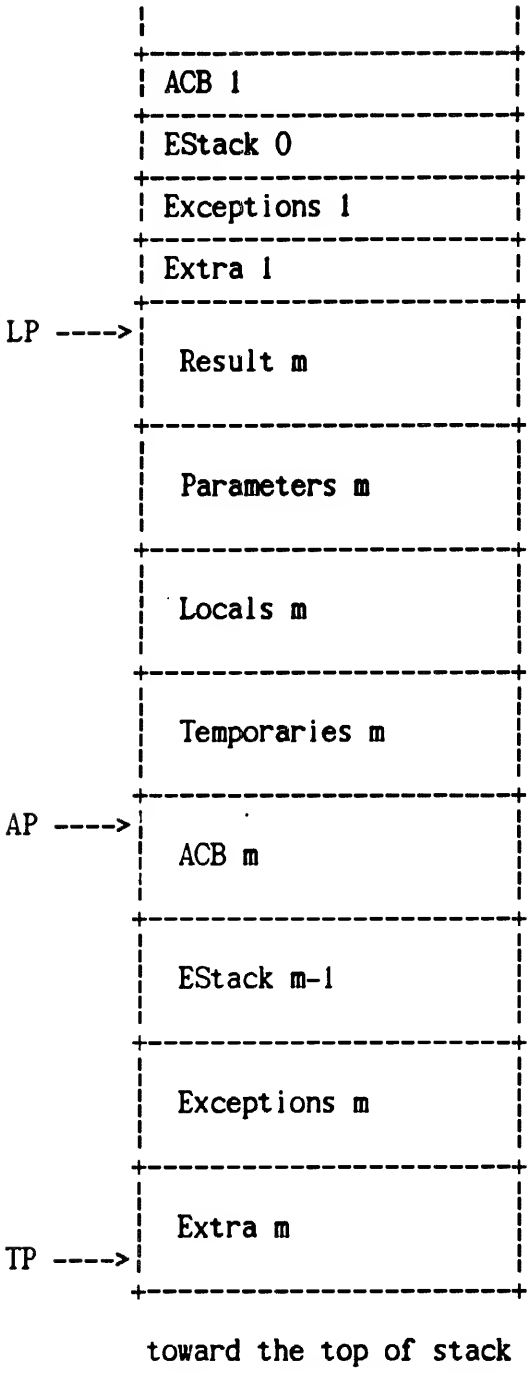
The EStack image immediately follows the ACB and looks like this:



1.B.1.c Run-Time Stack Organization

The following is an outline of the stack for a process of n segments, executing the mth routine call, which is in the ith segment:




ACB 1
EStack 0
Exceptions 1
Extra 1
LP ---->
Result m
Parameters m
Locals m
Temporaries m
AP ---->
ACB m
EStack m-1
Exceptions m
Extra m
TP ---->
toward the top of stack

1.B.2 Memory Organization at the System Level

The system makes use of two tables to control memory usage, the System Segment Address Table and the System Segment Information Table. The former contains all information which is needed by the Q-Code micro-code (location, size, residency, etc). The latter contains other information which is only referenced by the operating system (reference, I/O and lock counts; maximum size; etc).

1.B.2.a System Segment Address Table

The System Segment Address Table is a dynamic table, which is always resident in physical memory starting at physical address 0. This table contains two words per segment, and contains all information that the Q-Code micro-code needs to know about each segment. The information contained in this table is:

1) Segment Base Address (upper 12 bits)

2) Segment Size (number of 256 word blocks - 1)

3) Flags
   - Not Resident
   - Recently Used
   - Moving
   - Shareable
   - Segment Kind
   - Segment Full
   - Segment Table Entry In Use

The Segment Base Address is the upper 12 bits of the physical address of the base of the segment. If the segment is not resident in physical memory, this field is undefined. The lower 8 bits of the Segment Base Address are always guaranteed to be zero (since all segments are aligned on 256-word boundaries).

The Segment Size is one less than the size of the segment in 256-word blocks (i.e., Segment Size 0 = 256 words).

The Flags have the following meanings and uses:

Not Resident - When true, this flag indicates that the segment is either swapped out or that the segment table entry is not in use. When false, this flag indicates that the entry is in use and the segment it describes is resident in physical memory. (See the "Segment Table Entry In Use" flag.)

Recently Used - This flag is set when a segment is accessed. It is used by the swapper to determine which segments are likely candidates to be swapped out when space is needed.

Moving - This flag, when true, indicates that the segment is being moved from one location in physical memory to another. If moving is true, Resident will be false. Moving is used only by the swapper to determine how to handle segment faults. (Not used by the Q-Code micro-code).

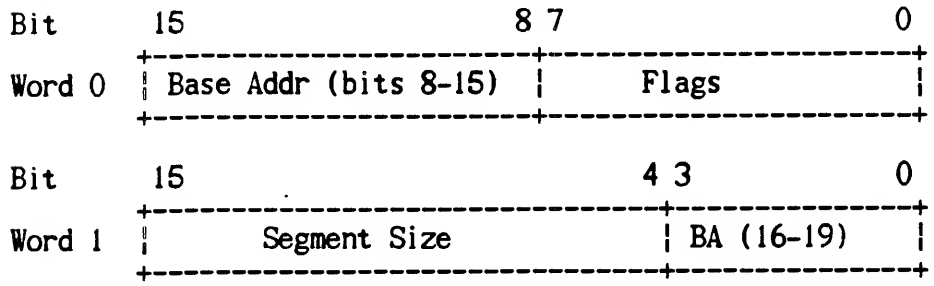
Shareable - When true, this flag indicates that a segment may be shared by several processes. (Not used by the Q-Code micro-code)

Segment Kind - This flag indicates whether the segment is a data or code segment. (Not used by the Q-Code micro-code)

Segment Full - This flag, when true, indicates that the entire data segment has been allocated (via the Pascal New procedure). This flag is needed to distinguish full and empty data segments (and has no relevant meaning for code segments). (Not used by the Q-Code micro-code)

Segment Table Entry In Use - This flag is set true when the segment table entry describes a valid segment.

The arrangement of these fields within the two words are shown below:



The positions of the flags within the low byte of Word 0 are:

Bit	Flag
0	Resident
1	Moving
2	Recently Used
3	Shareable
4	Segment Kind
5	Segment Full
6	Table Entry In Use
7	not used

1.B.2.b System Segment Information Table

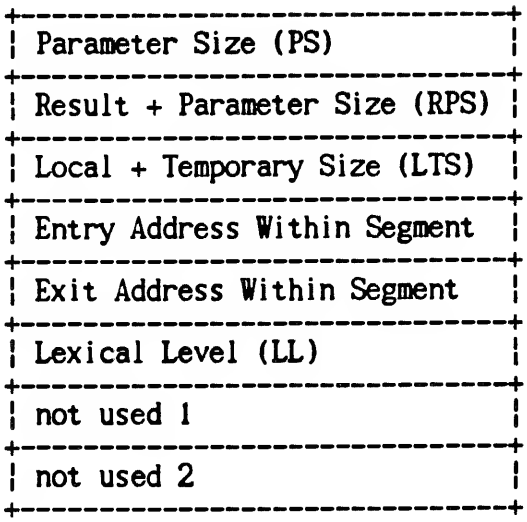
There is no information in the System Segment Information Table which is needed by the Q-Code micro-code; hence it is not described here. See "Module Memory" in the Operating System Manual.

1.B.2.c Code Segment Organization

A code segment contains the code for all routines in a segment and a routine dictionary which contains vital information about each of these routines.

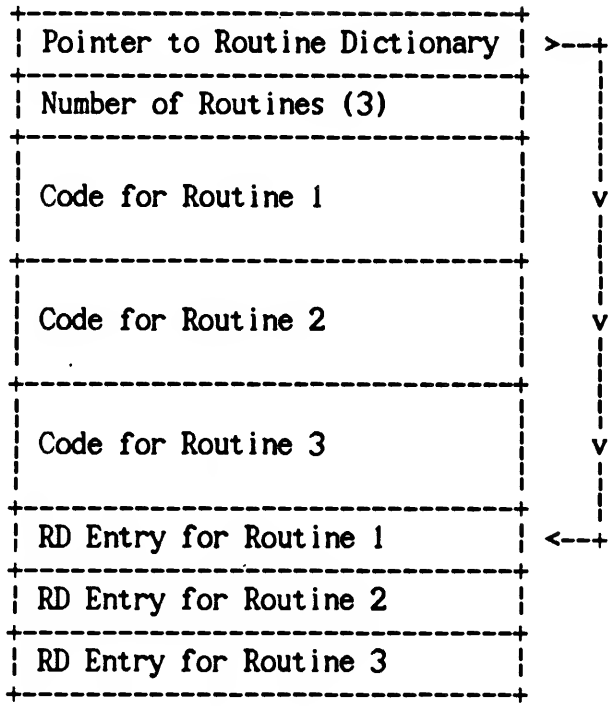
The first word of every code segment is the offset from the base of the segment to the first word of the routine dictionary. The second word contains the number of routines which are defined in the segment. These two words are followed by the actual code which comprise the routines. Finally, the code is followed by the routine dictionary. The code is padded with 0 to 3 words of zero (by the compiler) so that the routine dictionary is aligned on a quad-word boundary. This is possible since the compiler knows that the base of the segment is also aligned on a quad-word boundary. It should also be noted that each entry in the dictionary is exactly 2 quad-words long (8 words). The routine dictionary is indexed by (Base Address of Dictionary)+8*RN.

Each entry has the following form:



The Entry and Exit Addresses are the offsets from code base (CB) to the beginning of the routine and the beginning of the "terminate code" of the routine.

The following is a sample of a code segment containing 3 routines:



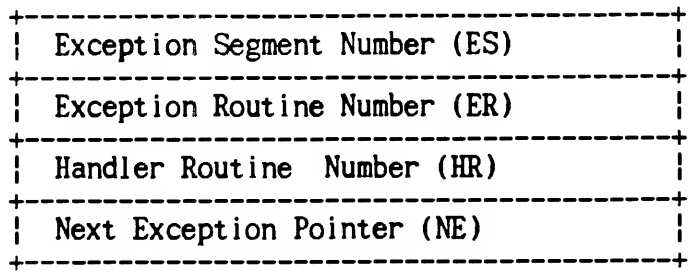
1.C Error Handling and Fault Conditions

Error processing is done through an exception handling mechanism. The syntax of exception and handler is described in the PERQ Pascal Extensions manual. The QCodes used to enable and raise exceptions are described in Section "Routine Calls and Returns."

An exception is declared in a way which is similar to a procedure declaration. Therefore it is convenient to assign a routine number at the point of definition of an exception. There is a corresponding entry in the routine dictionary to describe this exception. This entry describes a procedure with the correct number of parameters along with no locals or temporaries. The system segment number of the module, which contains the exception, and its routine number uniquely identify the exception.

An exception handler is simply a procedure--handlers may not return a value. An exception handler is enabled by declaring it inside some routine. This outer routine is called the enabler of the handler. The code segment numbers and global pointers of the enabler and handler are the same. The static link of the handler is the same as the activation pointer of the enabler. Thus an exception handler is uniquely identified by the ACB of the enabler and the routine number of the handler.

An exception enable record consists of the definition of the exception to be handled (ER and ES), the routine number of the handler (HR), and a link to the next exception enable record (NE). ER and ES are negative for a handler of all exceptions.



When a routine is called, the exception pointer (EP) within the new ACB is set to zero to indicate that there are no exception handlers. If exception handlers are declared within the routine, the compiler generates appropriate QCodes to add those handlers to the routine's exception list. When a routine is exited, the exception records are popped from the run-time stack along with the rest of the activation record.

When an exception is raised, the QCode interpreter microcode calls the procedure RaiseP in the module Except. This routine searches back through the run-time stack to find the most recent routine which contains a handler for that exception. Once such a candidate is found, the stack is searched again to determine if that handler is already active. If it is, the search for a candidate continues. This implementation ensures that while a particular instance of a handler is active, it will not be activated again. A recursive routine may contain a handler, and there may be several instances of the same handler. In this case, each handler is activated separately.

Determining if a handler is active by searching the stack is not the best method. If we assume that the depths of handler activation records are related to the number of active handlers, then the total stack search time for raising an exception is related to the square of the number of active handlers of that exception. This should rarely be burdensome because it is not expected that there will be more than one or two active handlers for a given exception. Recursive routines that pass exceptions up the call stack are pathological cases.

When an unused enable for the exception is found, the associated handler is called. The handler has the option of exiting to the routine which enabled the exception (via a Goto) or of returning to the point where the exception was raised (by falling off the end of the procedure).

If no handler is found, the default handler is called. The search order for a certain routine's exception list is the reverse of the order in which they were enabled. If a handler of all exceptions is declared within a certain routine, the compiler enables it first.

A handler of all exceptions is called in a special way. When it is called, the parameters that were passed when the exception was raised have already been pushed onto the stack. A new activation record for the handler is built above those parameters. Such a handler for all exceptions must have four words of parameters: the segment and routine numbers of the exception that was raised, a pointer to the first word of the original parameters, and a pointer to the first word after the original parameters. The two pointers are represented as integer offsets from the base of the stack.

2. Instruction Format

Instructions on the Q-machine are one byte long followed by zero to four parameters. Parameters are either a signed byte (B : range -128 to 127), an unsigned byte (UB : range 0 to 255) or a word (W). Words need not be word aligned (unless specified). The low byte is first in the instruction byte stream.

Any exceptions to these formats are noted with the instructions where they occur.

3. Pointers

There are five different types of pointers, defined as follows: (Note: 20-bit offsets may only exist on the EStack).

Word Pointer: A 20-bit offset from StackBase (StackBase is the 20 bit physical address of the base of the stack).

Byte Pointer: A 20-bit offset from StackBase to the base of the byte array (TOS-1) and a byte offset into the array (TOS).

String Pointer: Same as a byte pointer.

Packed Field Pointer: A 20-bit offset from StackBase to the base of the word the field is in (TOS-1) and a one word field descriptor (TOS).

Field Descriptor:
Bits 0-3: The field width (in bits) minus 1

Bits 4-7: The rightmost bit of the field.

Pascal Pointer: Obtained by declaring a variable as a pointer to another data type (i.e., var I: ^Integer;). (TOS-1) is the system segment number that contains the datum. (TOS) is the offset from the segment base to the datum.

Implementation Note: Stacks grow from low addresses to high addresses (i.e., if the address of TOS is 10 then the address of TOS-1 is 9 -- not 11).

4. QCode Descriptions

This section provides a detailed description for each QCode. The QCode descriptions appear categorically. The following lists the QCodes and equates each with its respective QCode number. Thus, if you only know the QCode number, you can use the list to equate the number to the named QCode.

QCode OpCode Definitions:

```
LDC0      = 0; (Assignment of Byte/Word opcodes are important)
LDC1      = 1;
LDC2      = 2;
LDC3      = 3;
LDC4      = 4;
LDC5      = 5;
LDC6      = 6;
LDC7      = 7;
LDC8      = 8;
LDC9      = 9;
LDC10     = 10;
LDC11     = 11;
LDC12     = 12;
LDC13     = 13;
LDC14     = 14;
LDC15     = 15;
LDCM0     = 16;
LDCB      = 17;
LDCW      = 18;
LSA       = 19;
ROTSHI    = 20;
STIND     = 21;
LDCN      = 22;
LDB       = 23;
STB       = 24;
LDCH      = 25;
LDP       = 26;
STP       = 27;
STCH      = 28;
EXG0      = 29;
LAND      = 30;
LOR       = 31;
LNOT      = 32;
EQUBool   = 33; (Opcode assignment of all EQU,NEQ,LEQ,LES)
NEQBool   = 34; (GEQ and GTR qcodes are important
LEQBool   = 35;
LESBool   = 36;
GEQBool   = 37;
GTRBool   = 38;
```

```
EQUI      = 39;
NEQI      = 40;
LEQI      = 41;
LESI      = 42;
GEQI      = 43;
GTRI      = 44;
EQUStr    = 51;
NEQStr    = 52;
LEQStr    = 53;
LESStr    = 54;
GEQStr    = 55;
GTRStr    = 56;
EQUByt    = 57;
NEQByt    = 58;
LEQByt    = 59;
LESByt    = 60;
GEQByt    = 61;
GTRByt    = 62;
EQUPowr   = 63;
NEQPowr   = 64;
LEQPowr   = 65;
SGS       = 66; (there is no LESPowr
GEQPowr   = 67;
SRS       = 68; (there is no GTRPowr
EQUWord   = 69; (Word is the last comparison and only EQU
NEQWord   = 70; and NEQ exist)
ABI       = 71;
ADI       = 72;
NGI       = 73;
SBI       = 74;
MPI       = 75;
DVI       = 76;
MODI      = 77;
CHK       = 78;
INN       = 88;
UNI       = 89;
QINT      = 90;
DIF       = 91;
EXIT      = 92;
NOOP      = 93;
REPL      = 94;
REPL2     = 95;
MMS       = 96;
MES       = 97;
LVRD      = 98;
LSSN      = 99;
XJP       = 100;
RASTEROP  = 102;
STARTIO   = 103;
INTOFF    = 105;
```

```
INTON    = 106;
LDLB     = 107;
LDLW     = 108;
LDL0     = 109;
LDL1     = 110;
LDL2     = 111;
LDL3     = 112;
LDL4     = 113;
LDL5     = 114;
LDL6     = 115;
LDL7     = 116;
LDL8     = 117;
LDL9     = 118;
LDL10    = 119;
LDL11    = 120;
LDL12    = 121;
LDL13    = 122;
LDL14    = 123;
LDL15    = 124;
LLAB     = 125;
LLAW     = 126;
STLB     = 127;
STLW     = 128;
STL0     = 129;
STL1     = 130;
STL2     = 131;
STL3     = 132;
STL4     = 133;
STL5     = 134;
STL6     = 135;
STL7     = 136;
LD0B     = 137;
LD0W     = 138;
LD00     = 139;
LD01     = 140;
LD02     = 141;
LD03     = 142;
LD04     = 143;
LD05     = 144;
LD06     = 145;
LD07     = 146;
LD08     = 147;
LD09     = 148;
LD010    = 149;
LD011    = 150;
LD012    = 151;
LD013    = 152;
LD014    = 153;
LD015    = 154;
LOAB     = 155;
```

```
LOAW     = 156;
STOB     = 157;
STOW     = 158;
STO0     = 159;
STO1     = 160;
STO2     = 161;
STO3     = 162;
STO4     = 163;
STO5     = 164;
STO6     = 165;
STO7     = 166;
MVBB     = 167;
MVBW     = 168;
MOVB     = 169;
MOVW     = 170;
INDB     = 171;
INDW     = 172;
SIND0    = 173;
SIND1    = 174;
SIND2    = 175;
SIND3    = 176;
SIND4    = 177;
SIND5    = 178;
SIND6    = 179;
SIND7    = 180;
LDIND    = 173; (Same as IND0))
LGAWW    = 181;
STMW     = 182;
STDW     = 183;
SAS      = 184;
ADJ      = 185;
CALL     = 186;
CALLV    = 187;
ATPB     = 188;
ATPW     = 189;
WCS      = 190;
JCS      = 191;
LDGB     = 192;
LDGW     = 193;
LGAB     = 194;
LGAW     = 195;
STGB     = 196;
STGW     = 197;
RETURN   = 200;
MMS2     = 201;
MES2     = 202;
LDTP     = 203;
JMPB     = 204;
JMPW     = 205;
JFB      = 206;
```

```
JFW       = 207;
JTB       = 208;
JTW       = 209;
JEQB      = 210;
JEQW      = 211;
JNEB      = 212;
JNEW      = 213;
IXP       = 214;
LDIB      = 215;
LDIW      = 216;
LIAB      = 217;
LIAW      = 218;
STIB      = 219;
STIW      = 220;
IXAB      = 221;
IXAW      = 222;
IXA1      = 223;
IXA2      = 224;
IXA3      = 225;
IXA4      = 226;
TLATE1    = 227;
TLATE2    = 228;
TLATE3    = 229;
EXCH      = 230;
EXCH2     = 231;
INCB      = 232;
INCW      = 233;
CALLXB    = 234;
CALLXW    = 235;
LDMC      = 236;
LDDC      = 237;
LDMW      = 238;
LDDW      = 239;
STLATE    = 240;
LINE      = 241;
ENABLE    = 242;
RAISE     = 243;
LDAP      = 244;
ROPS      = 250; (See below for 2nd byte)
INCDDS    = 251;
LOPS      = 252; (See below for 2nd byte)
BREAK     = 254;
ReFillOp  = 255;
```

Real Operations - Second byte of ROPS opcode:

```
TNC       = 0;
FLT       = 1;
ADR       = 2;
NGR       = 3;
```

```
SBR       = 4;
MPR       = 5;
DVR       = 6;
RND       = 7;
ABR       = 8;
EQUReal   = 9;
NEQReal   = 10;
LEQReal   = 11;
LESReal   = 12;
GEQReal   = 13;
GTRReal   = 14;
RUNUSED   = 15;
```

Long Operations - Second byte of LOPS opcode:

```
CVTLI     =  0;
CVTIL     =  1;
ADL       =  2;
NGL       =  3;
SBL       =  4;
MPL       =  5;
DVL       =  6;
MODL      =  7;
ABL       =  8;
EQULong   =  9;
NEQLong   = 10;
LEQLong   = 11;
LESLong   = 12;
GEQLong   = 13;
GTRLong   = 14;
LUnused   = 15;
```

4.A Variable Fetching, Indexing, Storing and Transferring

4.A.1 Loads and Stores of One Word

4.A.1.a Constant One Word Loads

LDC0..15	0-15	Load Word Constant. Pushes the value (0..15), with high byte zero, onto the EStack.
LDCN	22	Load Constant Nil. Pushes the value of NIL onto the EStack.
LDCM0	16	Load Constant -1.

LDCB	B	17	Load Constant Byte. Pushes the next byte on the EStack, with sign extend.
LDCW	W	18	Load Constant Word. Pushes the next word on the EStack.

4.A.1.b Local One Word Loads and Stores

LDL0..15 109-124 Short Load Local Word. LDLx fetches the word with offset x in the current activation record and pushes it onto the EStack.

LDLB UB 107 Load Local Word/Byte Offset. Fetches the word with offset UB in the current activation record and pushes it on the EStack.

LDLW W 108 Load Local Word/Word Offset. Fetches the word with offset W in the current activation record and pushes it on the EStack.

LLAB UB 125 Load Local Address/Byte Offset. Pushes a word pointer to the word with offset UB in the current activation record on EStack.

LLAW W 126 Load Local Address/Word Offset. Pushes a word pointer to the word with offset W in the current activation record on EStack.

STL0..7 129-136 Short Store Local Word. Store (ETOS) into word with offset x in the current activation record.

STLB UB 127 Store Local Word/Byte Offset. Store (ETOS) into word with offset UB in the current activation record.

STLW W 128 Store Local Word/Word Offset. Store (ETOS) into word with offset W in the current activation record.

Implementation Note: The address of the first local (offset 0) is contained in the Local Pointer register (LP). The address of the Nth local is computed as (LP) + N.

4.A.1.c Own One Word Loads and Stores

LDO0..15		139-154	Short Load Own Word. LDOx fetches the word with offset x in the current Global Data Block (GDB) and pushes it on the EStack.
LDOB	UB	137	Load Own Word/Byte Offset. Fetches the word with offset UB in the current Global Data Block (GDB) and pushes it on the EStack.
LDOW	W	138	Load Own Word/Word Offset. Fetches the word with offset W in the current Global Data Block (GDB) and pushes it on the EStack.
LOAB	UB	155	Load Own Address/Byte Offset. Pushes a word pointer to the word with offset UB in the current Global Data Block (GDB) on EStack.
LOAW	W	156	Load Own Address/Word Offset. Pushes a word pointer to the word with offset W in BASE activation record on EStack.
STO0..7		159-166	Short Store Own Word. STOx stores (ETOS) into the word with offset x in the current Global Data Block (GDB).
STOB	UB	157	Store Own Word/Byte Offset. Stores (ETOS) into the word with offset UB in the current Global Data Block (GDB).
STOW	W	158	Store Own Word/Word Offset. Stores (ETOS) into the word with offset W in the current Global Data Block (GDB).

Implementation Note: The address of the first own (offset 0) is contained in the Global Pointer register (GP). The address of the Nth own is computed as (GP)+N.

4.A.1.d Global One Word Loads and Stores

LDGB UB1,UB2 192 Load Global Word/Byte Offset. Loads the word with offset UB2 in the Global Data Block (GDB) for program segment UB1 onto EStack.

LDGW UB,W 193 Load Global Word/Word Offset. Same as LDGB except a full word offset is used.

LGAB UB1,UB2 194 Load Global Address/Byte Offset. Pushes a word pointer to the word with offset UB2 in the Global Data Block (GDB) for program segment UB1 onto EStack.

LGAW UB,W 195 Load Global Address/Word Offset. Same as LGAB except a full word offset is used.

LGAWW W1,W2 181 Load Global Address/Word Segment, Word Offset. Same as LGAB except a full word is used both for the segment number and the offset.

STGB UB1,UB2 196 Store Global Word/Byte Offset. Stores (ETOS) in word with offset UB2 in the Global Data Block (GDB) for program segment UB1.

STGW UB,W 197 Store Global Word/Word Offset. Same as STGB except a full word offset is used.

Note: To achieve LDGW and STGW with full word segment numbers, use LGAWW with LDIND or STIND.

Implementation Note: Self-relative pointers to the Global Data Blocks (GDB) for each externally referenced segment are contained in the External Segment Table (XST), pointed to by the Global Pointer (GP). The address of the first global (offset 0) in the designated GDB is computed as GP - 2 * ISN, where ISN (Internal Segment Number) is the program segment number specified in the load or store instruction. The Nth global is addressed by the base address (computes as above) plus N.

4.A.1.e Intermediate One Word Loads and Stores

LDIB	UB1,UB2	215	Load Intermediate Word/Byte Offset. UB1 indicates the number of static links to traverse to find the activation record to use. UB2 is the offset within the activation record of the desired word. The datum is pushed on EStack.
LDIW	UB,W	216	Load Intermediate Word/Word Offset. Same as LDIB except a word offset is used.
LIAB	UB1,UB2	217	Load Intermediate Address/Byte Offset. A word pointer is pushed on EStack (determined as in LDIB).
LIAW	UB,W	218	Load Intermediate Address/Word Offset. A word pointer is pushed on EStack (determined as in LDIW).
STIB	UB1,UB2	219	Store Intermediate Word/Byte Offset. Stores (ETOS) in memory (address determined as in LDIB).
STIW	UB,W	220	Store Intermediate Word/Word Offset. Stores (ETOS) in memory (address determined as in LDIW).

Implementation Note: The Activation Pointer register (AP) contains the address of the current Activation Control Block (ACB). Within the ACB is the Static Link (SL) to the previous ACB. To compute the address of the first intermediate word of the desired level, traverse the Static Links to the correct ACB. Within the ACB is the Local Pointer (LP) for that activation record.

4.A.1.f Indirect One Word Loads and Stores

STIND 21 Store Indirect. (ETOS) is stored into the word pointed to by word pointer (ETOS-1).

LDIND 173 Load Indirect. Word pointed to by word pointer (ETOS) is pushed on EStack.

4.A.2 Loads and Stores of Multiple Words

4.A.2.a Double Word Loads and Stores (Reals and Pointers)

LDDC <block> 237 Load Double Word Constant. <block> is a double word constant. Load the constant onto EStack.

LDDW 239 Load Double Word. (ETOS) is a word pointer to a double word. The double word is pushed onto EStack.

STDW 183 Store Double Word. (ETOS),(ETOS-1) is a double word and (ETOS-2) is a word pointer to a double word block of memory. The double word is popped from ESTACK into the double word pointed to by (ETOS-2).

4.A.2.b Multiple Word Loads and Stores (Sets)

LDMC UB,<block> 236 Load Multiple Word Constant. UB is the number of words to load, and <block> is a block of UB words, in reverse word order. Load the block onto the MStack.

LDMW 238 Load Multiple words. (ETOS-1) is a word pointer to the beginning of a block of (ETOS) words. Push the block onto the MStack.

STMW 182 Store Multiple Words. The MStack contains a block of (ETOS) words, (ETOS-1) is a word pointer to a similar block. Transfer the block from MStack to the destination block.

4.A.3 Byte Arrays

Note: A byte pointer is loaded onto the stack with a LLA, LOA or LGA of the base address of the array followed by the computation of the offset.

LDB	23	Load Byte. Push the byte (after zeroing the high Byte) pointed to by byte pointer (ETOS),(ETOS-1) on EStack.
STB	24	Store Byte. Store the low byte of (ETOS) into the location specified by byte pointer (ETOS-1),(ETOS-2).
MVBB UB	167	Move Bytes/Byte Counter. (ETOS),(ETOS-1) is a source byte pointer to a block of UB bytes, and (ETOS-2),(ETOS-3) is the destination byte pointer to a similar block. Transfer the source block to the destination block.
MVBW	168	Move Bytes/Word Counter. Same as MVBB except (ETOS-1), (ETOS-2) is the source byte pointer, (ETOS-3), (ETOS-4) is the destination byte pointer, and (ETOS) is the number of bytes to transfer.

4.A.4 Strings

LSA UB,<chars> 19 Load String Address. UB is the length of the string constant <chars>. A string pointer is pushed on EStack (the virtual address of UB is pushed followed by a zero). UB is word aligned.

SAS 184 String Assign. (ETOS-1),(ETOS-2) is the source string pointer, and (ETOS-3),(ETOS-4) is the destination string pointer. (ETOS) is the declared length of the destination. The length of the source and destination are compared, and if the source string is longer than the destination, a run-time error occurs. Otherwise all bytes of source containing valid information are transferred to the destination string.

LDCH 25 Load Character. (ETOS),(ETOS-1) is a string pointer. (ETOS) is checked to insure that it lies within the dynamic length of the string. If so, the character pointed to by (ETOS),(ETOS-1) is pushed; otherwise, a run-time error occurs.

STCH 28 Store Character. (ETOS) is a character and (ETOS-1),(ETOS-2) is a string pointer. (ETOS-1) is checked to insure that it lies within the dynamic length of the string. If so, the character (ETOS) is stored in the string, at the position pointed to by (ETOS-1),(ETOS-2); otherwise, a run-time error occurs.

4.A.5 Record and Array Indexing and Assignment

MOVB UB 169 Move Words/Byte Counter. (ETOS) is a word pointer to a block of UB words, and (ETOS-1) is a word pointer to a similar block. The block pointed to by (ETOS) is transferred to the block pointed to by (ETOS-1).

MOVW 170 Move Words/Word Counter. Same as MOVB except (ETOS-1) is the source pointer, (ETOS-2) is the destination pointer, and (ETOS) is the number of words to be transferred.

SIND0-7 173-180 Short Index and Load Word. SINDx indexes the word pointer (ETOS) by x words, and pushes the word pointed to by the result on ESTACK. (Note: SIND0 is synonymous to LDIND).

INDB UB 171 Static Index and Load Word/Byte Index. Indexes the word pointer (ETOS) by UB words, and pushes the word pointed to by the result on ESTACK.

INDW W 172 Static Index and Load Word/Word Index. Same as INDB except a full word index is used.

INCB UB 232 Increment Field Pointer/Byte Index. The word pointer (ETOS) is indexed by UB words and the resultant pointer is pushed on ESTACK.

INCW W 233 Increment Field Pointer/Word Index. Same as INCB except a full word index is used.

Note: INCB and INCW are equivalent to add UB or W to (ETOS).

IXAB UB 221 Index Array/Byte Array Size. (ETOS) is an integer index, (ETOS-1) is a word pointer to the base of the array, and UB is the size (in words) of an array element. A word pointer to the first word of the indexed element is pushed on ESTACK.

IXAW 222 Index Array/Word Array Size. Same as IXAB except (ETOS-1) is the integer index, (ETOS-2) is the word pointer to the base of the array, and (ETOS) is the size (in words) of an array element.

IXA1..4 223-226 Index Array/Short Array Size. Same as IXAB except array element sizes are fixed at 1-4.

IXP UB 214 Index Packed Array. (ETOS) is an integer index, and (ETOS-1) is a word pointer the base of the array. Bits 4-7 of UB contain the number of elements per word minus 1, and bits 0-3 contain the field width (in bits) minus 1. Compute and push a packed field pointer.

LDP 26 Load a Packed Field. Push the field described by the packed field pointer (ETOS),(ETOS-1) on ESTACK.

STP 27 Store into Packed Field. Store (ETOS) in the field described by the packed field pointer (ETOS-1),(ETOS-2).

ROTSHI UB 20 Rotate/Shift. (ETOS-1) is the argument to be rotated or shifted, and (ETOS) is the distance to rotate or shift. If UB is 0 then a right rotate occurs, and if UB is 1 then a shift occurs. The direction of the shift is determined from (ETOS); If (ETOS) >= 0 then a left shift occurs; otherwise, a right shift. (ETOS) must be in the range from -15 to +15.

4.B Top of Stack Arithmetic and Comparisons

4.B.1 Logical

LAND	30	Logical Add. AND (ETOS) into (ETOS-1).
LOR	31	Logical Or. OR (ETOS) into (ETOS-1).
LNOT	32	Logical Not. Take one's complement of (ETOS).
EQUBOOL	33	Boolean =,
NEQBOOL	34	<>,
LEQBOOL	35	<=,
LESBOOL	36	<,
GEQBOOL	37	>=,
GTRBOOL	38	and > comparisons. Compare (ETOS-1) to (ETOS) and push true or false on ESTACK.

4.B.2 Integer

ABI	71	Absolute Value of Integer. Take absolute value of (ETOS). Result is undefined if (ETOS) is initially -32768.
ADI	72	Add Integers. Add (ETOS) and (ETOS-1).
NGI	73	Negate Integer. Take the twos complement of (ETOS).
SBI	74	Subtract Integers. Subtract (ETOS) from (ETOS-1).
MPI	75	Multiply Integers. Multiply (ETOS) and (ETOS-1). This instruction may cause overflow if the result is larger than 16 bits.
DVI	76	Divide Integers. Divide (ETOS-1) by (ETOS) and push quotient (as defined by Jensen and Wirth).
MODI	77	Modulo Integers. Divide (ETOS-1) by (ETOS) and push the remainder (as defined by Jensen and Wirth).
CHK	78	Check Against Subrange Bounds. Insure that (ETOS-1) <= (ETOS-2) <= (ETOS), leaving (ETOS-2) on top of the stack. If conditions are not met a run-time error occurs.
EQUI	39	Integer =,
NEQI	40	<>,
LEQI	41	<=,
LESI	42	<,
GEQI	43	>=,
GTRI	44	and > comparisons. Compare (ETOS-1) to (ETOS) and push true or false on ESTACK.

4.B.3 Real Operations

ROPS UB 250 Real Operations. Arithmetic operations on floating point (32 bit) values. In general, (ETOS) is the low-order word and (ETOS-1) is the high-order word of the value. When two floating point values are involved, (ETOS-2) is the low-order word of the second real and (ETOS-3) is the high-order word.

All over/underflows cause a run-time error. Division by zero (0) also causes a run-time error.

UB determines the operation according to the following table:

0 - The real (ETOS),(ETOS-1) is truncated (as defined by Jensen and Wirth), converted to a single precision integer, and pushed onto EStack.

1 - The single precision integer (ETOS) is converted to a floating-point number and pushed onto EStack.

2 - Add (ETOS),(ETOS-1) and (ETOS-2),(ETOS-3).

3 - Negate the real (ETOS),(ETOS-1).

4 - Subtract (ETOS),(ETOS-1) from (ETOS-2),(ETOS-3).

5 - Multiply (ETOS),(ETOS-1) and (ETOS-2),(ETOS-3).

6 - Divide (ETOS),(ETOS-1) by (ETOS-2),(ETOS-3).

7 - The real (ETOS),(ETOS-1) is rounded (as defined by Jensen and Wirth), truncated and converted to a single precision integer, and pushed onto EStack.

8 - Take the absolute value of the real (ETOS),(ETOS-1).

9 - = of two real values. (ETOS) = true or false.

10 - <> of two real values.

11 - <= of two real values.

12 - < of two real values.

13 - >= of two real values.

14 - > of two real values.

LEQPOWR	65	<= (subset of),
GEQPOWR	67	and >= (superset of) comparisons of the two sets on top of ESTACK, with sizes (ETOS) and (ETOS-1).

4.B.5 Strings

EQUSTR	51	String =,
NEQSTR	52	<>,
LEQSTR	53	<=,
LESSTR	54	<,
GEQSTR	55	>=,
GTRSTR	56	and > comparisons. The string pointed to by string pointer (ETOS-2),(ETOS-3) is lexicographically compared to the string pointed to by string pointer (ETOS),(ETOS-1).

4.B.6 Byte Arrays

EQUBYT	UB	57	Byte Array =,
NEQBYT	UB	58	<>,
LEQBYT	UB	59	<=,
LESBYT	UB	60	<,
GEQBYT	UB	61	>=,
GTRBYT	UB	62	and >

comparisons. <=, <, >=, and > are only emitted for packed arrays of characters. The argument, UB, if non-zero, is the size of the array. If UB is equal to 0, then (ETOS) is the size of the array.

4.B.7 Array and Record Comparisons

EQUWORD UB 69 Word or multiword structure =

NEQWORD UB 70 and <> comparisons. The argument, UB, if non-zero, is the size of the array. If UB equals 0, then (ETOS) is the size of the array.

4.B.8 Long Operations

LOPS UB 252 Long Operations. Arithmetic operations on long (32 bit) values. In general, (ETOS) is the low-order word and (ETOS-1) is the high-order word of the value. When two long values are involved, (ETOS-2) is the low-order word of the second long and (ETOS-3) is the high-order word. UB determines the operation according to the following table:

0 - Converts the long value (ETOS), (ETOS-1) to a single word. The high- order word must be 0 or all 1's, as it is truncated. If not, a runtime error is generated.

1 - Converts a single word (ETOS) into a long value.

2 - Adds two long values.

3 - Negates long value.

4 - Subtracts two long values.

5 - Multiplies two long values.

6 - Divides two long values.

7 - Mods two long values.

8 - Absolute value of a long value.

9 - = of two long values. (ETOS) = true or false.

10 - <> of two long values.

11 - <= of two long values.

12 - < of two long values.

13 - >= of two long values.

14 - > of two long values.

4.C Jumps

JMPB B 204 Unconditional Jump/Byte Offset. B is added to the IPC. Negative values of B cause backward jumps.

JMPW W 205 Unconditional Jump/Word Offset. W is added to the IPC. Negative values of W cause backward jumps.

JFB B 206 False Jump/Byte Offset. Jump (as in JMPB) if (ETOS) is false.

JFW W 207 False Jump/Word Offset. Jump (as in JMPW) if (ETOS) is false.

JTB B 208 True Jump/Byte Offset. Jump (as in JMPB) if (ETOS) is true.

JTW W 209 True Jump/Word Offset. Jump (as in JMPW) if (ETOS) is true.

JEQB B 210 Equal Jump/Byte Offset. Jump (as in JMPB) if integer (ETOS) equals (ETOS-1).

JEQW W 211 Equal Jump/Word Offset. Jump (as in JMPW) if integer (ETOS) equals (ETOS-1).

JNEB B 212 Not Equal Jump/Byte Offset. Jump (as in JMPB) if integer (ETOS) is not equal to (ETOS-1).

JNEW W 213 Not Equal Jump/Word Offset. Jump (as in JMPW) if integer (ETOS) is not equal to (ETOS-1).

XJP W1,W2,W3,<Case Table> 100

Case Jump. W1 is word-aligned, and is the minimum index of the table. W2 is the maximum index. W3 is the offset to the code to be executed if the case specified has no entry in the case table. The case table is W2 - W1 + 1 words long and contains offsets to the code to be executed for each case.

If (ETOS), the actual index, is not in the range W1..W2 then W3 is added to PC. Otherwise, (ETOS) - W1 is used as an index into the case table and the index entry is added to PC.

4.D Routine Calls and Returns

Note: There can be at most 256 routines in a segment.

CALL UB 186 Call Routine. Call routine UB, which is in the current segment.

CALLXB UB1,UB2 234 Call External Routine/Byte Segment. UB1 is the internal segment number (ISN) which contains the routine numbered UB2 to be called. First the ISN is translated to the correct SSN, and residency of that segment is checked. If the segment is resident, the call proceeds; if not, the PC is backed up so that the call will be re-executed, and a segment fault occurs. The second attempt is guaranteed to succeed, since the process is unable to resume execution until the segment SSN is resident.

CALLXW W,UB 235 Call External Routine/Word Segment. Same as CALLXB except the internal segment number (ISN) is given in a full word.

LVRD W,UB1,UB2 98 Load Variable Routine Descriptor. This Q-Code pushes a Variable Routine Descriptor on the EStack for the routine UB1 in segment ISN W, at lexical level UB2. The following values (which comprise a variable routine descriptor) are pushed: (ETOS) = System Segment Number (SSN); (ETOS-1) = Global Pointer, represented as an offset from SB; (ETOS-2) = Routine Number; and (ETOS-3) = Static Link (determined as if a call were actually performed to the routine here).

CALLV 187 Call Variable Routine. The ESTACK elements (ETOS) --- (ETOS-3) are a variable routine descriptor (as described above in LVRD). Residency of the segment are checked. If the segment is resident, the call is made as will CALL, except the GP and SL are taken from the variable routine descriptor; if not, a segment fault occurs as with CALLX.

RETURN 200 Return from Routine. Return from the current routine. If the routine was a function, the function value is left on the top of the MStack. Since the first word of a code segment is not code, but an offset to the

routine dictionary, if the RA which is being returned to is 0, the return is performed to the exit code of that routine. (This proves useful for the EXIT and EXGO Q-Codes described below).

EXIT W,UB 92 Exit from Routine. Exit from all routines up to and including the most recent invocation of the routine UB in ISN W. This is accomplished by setting the RAs in all the ACBs to 0, from the most recent through and including the first ACB which was created from an invocation the routine to be exited, and jumping to the exit code of the current routine.

EXGO W1,UB,W2 29 Exit and Goto. Exit from all routines up to, but not including, routine UB in ISN W1, and then jump to the instruction with offset W2 from CB. The implementation is similar to EXIT, except the last RA modified is loaded with W2.

ENABLE W,UB1,UB2 242 Enable Exception Handler. W and UB1 are the internal segment and routine numbers, respectively, of the exception being enabled. UB2 is the routine number of the handler. A new exception enable record is pushed (quad word aligned) onto the MStack and linked into the routine's current exception list.

RAISE W1,UB,W2 243 Raise Exception. W1 and UB1 are the internal segment and routine numbers, respectively, of the exception to be raised. W2 is the number of words of parameters that have already been pushed onto the memory stack. The exception is raised.

4.E Systems Programs Support Procedures

BREAK	254	Breakpoint QCode. Causes a Qcode level breakpoint to the microcode kernel (KRNL).
NOOP	93	No-Operation.
REPL	94	Replicate. Replicate (ETOS).
REPL2	95	Replicate Two. Replicate two top-of-estack words (i.e., first push original (ETOS-1), then push original (ETOS)).
MMS	96	Move to Memory Stack. Push (ETOS) onto MTOS (16-bit transfer).
MES	97	Move to Expression Stack. Push (MTOS) onto ETOS (16-bit transfer - top 4 bits are zeroed).
MMS2	201	Move Double to Memory Stack. Transfer the top two words from the EStack to the MStack. The order is reversed; old (ETOS) is (MTOS-1), (ETOS-1) is (MTOS).
MES2	202	Move Double to Expression Stack. Transfer the top two words from the MStack to the EStack. THe order is reversed; old (MTOS) is (ETOS-1), (MTOS-1) is (ETOS).
RASTER-OP	102	RasterOp. RasterOp is a special QCode which is used to manipulate blocks of memory of arbitrary sizes. It is especially useful for creating and modifying displays on the screen. RasterOp modifies a rectangular area (called the "destination") of arbitrary size (to the bit). The picture drawn into this rectangle is computed as a function of the previous contents of the destination and the contents of another rectangle of the same size called the "source". The functions performed to combine the two pictures are described below. RasterOp can be used on memory other than that used for the screen bitmap. There are two parameters that specify the areas of memory to be used for the source and destination: a pointer to the start of the memory block and the length (in words) of scanlines in the block. A scanline is one of

the elements that cross the block. On the screen, for example, a scanline is one of the horizontal lines with a length of 48 words. Within these regions, the positions of the source and destination rectangles are given as offsets from the pointer. Thus position (0,0) would be at the upper left corner of the region, and, for the screen, (767, 1023) would be the lower right.

The EStack must be arranged in the following order for RASTER-OP:

(ETOS-10) Function
(ETOS-9) Width
(ETOS-8) Height
(ETOS-7) Destination-X-Position
(ETOS-6) Destination-Y-Position
(ETOS-5) Destination-Area-Line-Length
(ETOS-4) Destination-Memory-Pointer
(ETOS-3) Source-X-Position
(ETOS-2) Source-Y-Position
(ETOS-1) Source-Area-Line-Length
(ETOS) Source-Memory-Pointer

The values on the stack are defined below:

"Function" defines how the source and the destination are to be combined to create the final picture stored at the destination. The RasterOp functions are as follows (Src represents the source and Dst the destination):

Function	Name	Action
0	RRpl	Dst gets Src
1	RNot	Dst gets NOT Src
2	RAnd	Dst gets Dst AND Src
3	RAndNot	Dst gets Dst AND NOT Src
4	ROr	Dst gets Dst OR Src
5	ROrNot	Dst gets Dst OR NOT Src
6	RXor	Dst gets Dst XOR Src
7	RXNor	Dst gets Dst XNOR Src

"Width" specifies the size in the horizontal ("x") direction of the source and destination rectangles (given in bits).

"Height" specifies the size in the vertical ("y") direction of the source and destination rectangles (given in scan lines).

"Destination-X-Position" is the bit offset of the left side of the destination rectangle. The value is offset from Destination-Memory-Pointer (see below).

"Destination-Y-Position" is the scan-line offset of the top of the destination rectangle. The value is offset from Destination-Memory-Pointer (see below).

"Destination-Area-Line-Length" is the number of words which comprise a line in the destination region (hence defining the region's width). The appropriate value to use when operating on the screen is 48. The specified value must be a multiple of four (4) and within the range 4 through 48.

"Destination-Memory-Pointer" is the virtual address of the top left corner of the destination region. This pointer MUST be quad-word aligned, however.

"Source-X-Position" is the bit offset of the left side of the source rectangle. The value is offset from Source-Memory-Pointer (see below).

"Source-Y-Position" is the scan-line offset of the top of the source rectangle. The value is offset from Source-Memory-Pointer (see below).

"Source-Area-Line-Length" is the number of words which comprise a line in the source region (hence defining the region's width). The appropriate value to use when operating on the screen is 48. The specified value must be a multiple of four (4) and within the range 4 through 48.

"Source-Memory-Pointer" is the virtual address of the top left corner of the source region. This pointer MUST be quad-word aligned, however.

LINE	241	Line Drawing. (ETOS) is a pointer to the origin (relative 0,0) of the area on which the line is drawn. (ETOS-4) and (ETOS-3) are the x and y coordinates (respectively) of the first endpoint of the line. (ETOS-2) and (ETOS-1) are the x and y coordinates (respectively) of the second endpoints on the line. (ETOS-5) is the style of the line where a value of 1 means erase the line, 2 means to xor the line and anything else means to draw the line.
STARTIO	103	(ETOS) is the channel on which to start IO.
INTOFF	105	Disable interrupts.
INTON	106	Enable interrupts.
EXCH	230	Exchange. (ETOS) and (ETOS-1) are swapped.
EXCH2	231	Exchange Double. The pair (ETOS) and (ETOS-1) are swapped with the pair (ETOS-2) and (ETOS-3).
TLATE1	227	Translate Top of Stack. (ETOS),(ETOS-1) is a virtual address. If the segment SSN (ETOS-1) is resident, convert the virtual address to an offset from stack base (SB) and execute the next Q-Code (what ever it may be), with out interrupts, to competion. If the segment SSN (ETOS-1) is non-resident, restore the EStack to its previous state, backup the PC to re-execute the TLATE1 and perform a segment fault.
TLATE2	228	Translate Top of Stack - 1. Same as TLATE1 except the virtual address is at (ETOS-1),(ETOS-2).
TLATE3	229	Translate Top of Stack - 2. Same as TLATE1 except the virtual address is at (ETOS-2),(ETOS-3).

STLATE	UB	240	Special Translate. This translate is similar to the previous translate Q-Codes, except that it can specify a greater depth than TLATE3, and that it may specify the translation of 2 virtual addresses. Each half of UB is interpretted as the depth of the System Segment Number word of the virtual address to be translated (prior to any stack alteration). A depth of 0 indicates no translation. All segments specified in the STLATE must be resident before any translations occur; otherwise a segment fault occurs. Note, if both nibbles of UB are non-zero then the low-order nibble (bits 0-3) must be less than the high-order nibble (bits 4-7).
LSSN		99	Load Stack Segment Number. Pushes the system segment number of the MStack onto EStack.
LDTP		203	Load Top Pointer (plus 1). Pushes the value of Top Pointer (TP) plus 1 onto EStack.
LDAP		244	Load Activation Pointer. The current activation pointer (as an offset from the base of the stack) is pushed onto the EStack.
ATPB	SB	188	Add to Top Pointer/Byte Value. Adds SB to TP.
ATPW		189	Add to Top Pointer/Word Value. Adds (ETOS) to TP.
WCS		190	Write Control Store. A control store word is written from information on the EStack. (ETOS) is the address (with bytes exchanged) in the control store to which the word will be written. (ETOS-1) is the value to be written into the high-order third, (ETOS-2) is the value to be written into the middle third and (ETOS-3) is to be written into the low-order third.
JCS		191	Jump to a Location in the Control Store. Control is transferred to the control store address (with bytes exchanged) given in (ETOS). A routine called with JCS should exit with a NextInst(0) jump.

REFILLOP	255	Refill the OpFile. This instruction causes execution to proceed from the beginning of the next quad-word.
INCDDS	251	Increment Diagnostic Display. The value of the diagnostic display is incremented and the contents of the EStack is checked. If the EStack is not empty, a runtime error is generated.

INDEX